jardin Documentation

Release 0.19.8

Emmanuel Turlay

Sep 28, 2021

Contents

jardin *(noun, french)* – garden, yard, grove.

Jardin is a `pandas.DataFrame`-based ORM for Python applications.

CHAPTER 1

Getting started

1.1 Installation

```
$ pip install jardin
```

or

```
$ echo 'jardin' >> requirements.txt
$ pip install -r requirements.txt
```

1.2 Setup

In your working directory (the root of your app), create a file named `jardin_conf.py`:

```
# jardin_conf.py

DATABASES = {
  'my_master_database': 'https://username:password@master_database.url:port',
  'my_replica_database': 'https://username:password@replica_database.url:port'
}

LOG_LEVEL = logging.DEBUG

WATERMARK = 'My Great App'
```

You can also place this file anywhere you want and point to it with the environment variable `JARDIN_CONF`.

If you'd like to balance the load among a few databases - especially among replica databases - you may give multiple database URLs, separated by whitespace:

```
# jardin_conf.py

DATABASES = {
  'my_replicas': 'https://user:pass@replica1.url:port https://user:pass@replica2.
↪url:port'
}

# On first access, jardin randomly picks an URL from the list and maintains connection
# "stickiness" during the lifetime of the process. In a long-running process,
# application may ask jardin to switch to other connections on the list by
# calling 'jardin.reset_session()'.
```

Then, in your app, say you have a table called `users`:

```
# app.py
import jardin

class User(jardin.Model):
  db_names = {'master': 'my_master_database', 'replica': 'my_replica_database'}
```

In the console:

```
>>> from app import User
>>> users = User.last(4)
# /* My Great App */ SELECT * FROM users ORDER BY u.created_at DESC LIMIT 4;
>>> users
id   name    email              ...
0    John    john@beatl.es      ...
1    Paul    paul@beatl.es      ...
2    George  george@beatl.es    ...
3    Ringo   ringo@beatl.es     ...
```

The resulting object is a pandas dataframe:

```
>>> import pandas
>>> isinstance(users, pandas.DataFrame)
True
>>> isinstance(users, jardin.Collection)
True
```

CHAPTER 2

Querying

2.1 SELECT queries

Here is the basic syntax to select records from the database

```
>>> users = User.select(
              select=['id', 'name'],
              where={'email': 'paul@beatl.es'},
              order='id ASC',
              limit=1)
# SELECT u.id, u.name FROM users u WHERE u.email = 'paul@beatl.es' ORDER BY u.id ASC␣
↪LIMIT 1; /* My Great App */
>>> users
id   name
1    Paul
```

2.1.1 Arguments

See *API reference*.

where argument

Here are the different ways to feed a condition clause to a query.

- `where = "name = 'John'"`
- `where = {'name': 'John'}`
- `where = {'id': (0, 3)}` – selects where `id` is between 0 and 3
- `where = {'id': [0, 1, 2]}` – selects where `id` is in the array
- `where = [{'id': (0, 10), 'instrument': 'drums'}, ["created_at > %(created_at)s", {'created_at': '1963-03-22'}]]`

For other operators than =, see *Comparators*.

inner_join, left_join arguments

The simplest way to join another table is as follows

```
>>> User.select(inner_join=["instruments i ON i.id = u.instrument_id"])
```

If you have configured your models associations, see *Features*, you can simply pass the class as argument:

```
>>> User.select(inner_join=[Instrument])
```

2.1.2 Individual record selection

You can also look-up a single record by id:

```
>>> User.find(1)
# SELECT * FROM users u WHERE u.id = 1 LIMIT 1;
User(id=1, name='Paul', email='paul@beatl.es', ...)
>>> User.find_by(values={'name': 'Paul'})
# SELECT * FROM users u WHERE u.name = 'Paul' LIMIT 1;
User(id=1, name='Paul', email='paul@beatl.es', ...)
```

Note that the returned object is a `Record` object which allows you to access attributes in those way:

```
>>> user['name']
Paul
>>> user.name
Paul
```

2.2 INSERT queries

```
>>> user = User.insert(values={'name': 'Pete', 'email': 'pete@beatl.es'})
# INSERT INTO users (name, email) VALUES ('Pete', 'pete@beatl.es') RETURNING id;
# SELECT u.* FROM users WHERE u.id = 4;
>>> user
id   name    email
4    Pete    pete@beatl.es
```

2.3 UPDATE queries

```
>>> users = User.update(values={'hair': 'long'}, where={'name': 'John'})
# UPDATE users u SET (u.hair) = ('long') WHERE u.name = 'John' RETURNING id;
# SELECT * FROM users u WHERE u.name = 'John';
```

2.4 DELETE queries

```
>>> User.delete(where={'id': 1})
# DELETE FROM users u WHERE u.id = 1;
```

2.5 Raw queries

```
>>> jardin.query(sql='SELECT * FROM users WHERE id IN %(ids)s;', params={'ids': [1, 2,
↪ 3]})
# SELECT * FROM users WHERE id IN (1, 2, 3);
```

2.6 Query from SQL file

```
>>> jardin.query(filename='path/to/file.sql', params={...})
```

The path is relative to the working directory (i.e. where your app was launched).

CHAPTER 3

Comparators

The syntax `where={'id': 123}` works well for = conditions but breaks down for other operators. For that purpose, jardin offers comparators.

For example

```
>>> from jardin.comparators import *
>>> User.count(where={'created_at': gt(datetime.utcnow() - timedelta(day=1))})
# SELECT COUNT(*) FROM users WHERE created_at > '2018-04-29 12:00:00';
```

3.1 All comparators

Comparator	Operator	Example	Result
`lt`	<	`{'n': lt(3)}`	`WHERE n < 3`
`leq`	<=	`{'n': leq(3)}`	`WHERE n <= 3`
`gt`	>	`{'n': gt(3)}`	`WHERE n > 3`
`geq`	<=	`{'n': geq(3)}`	`WHERE n >= 3`
`not_null`		`{'n': not_null()}`	`WHERE n IS NOT NULL`
`not_in`		`{'n': not_in([1, 2])}`	`WHERE n IS NOT IN (1, 2)`

CHAPTER 4

Features

..Associations ..————

..Belongs-to and has-many relationships can be declared as such:

..And then used as such:

..Or:

4.1 Query watermarking

By defining a watermark in your `jardin_conf.py` file:

```
WATERMARK = 'MyGreatApp'
```

Queries will show up as such in your SQL logs:

```
/* MyGreatApp | path/to/file.py:function_name:line_number */ SELECT * FROM ....;
```

4.2 Scopes

Query scopes can be defined inside your model as such:

```
class User(jardin.Model):

  scopes = {
    'active': {'active': True},
    'recent': ["last_sign_up_at > %(week_ago)s", {'week_ago': datetime.utcnow() -␣
↪timedelta(weeks=1)}]
  }
```

Then used as such:

```
User.select(scopes = ['active', 'recent'])
```

Which will issue this statement

```
SELECT * FROM users u WHERE u.active IS TRUE AND u.last_sign_up_at > ...;
```

4.3 Soft deletes

If you don't want to actually remove rows from the database when deleting a record, you can activate soft-deletes:

```
class User(jardin.Model):

  soft_delete = True
```

When the `destroy` method is called on a model instance, the `deleted_at` database field on the corresponding table will be set to the current UTC time.

Then, when calling `select`, `count`, `delete` or `update`, rows with a non-NULL *deleted_at* value will be ignored. This can be overridden by passing the `skip_soft_delete=True` argument.

The `find` method is not affected.

To force delete a single record, call `destroy(force=True)`.

To customize the database column used to store the deletion timestamp, do:

```
class User(jardin.Model):

  soft_delete = 'my_custom_db_column'
```

4.4 Multiple databases and master/replica split

Multiple databases can be declared in `jardin_conf.py`:

```
DATABASES = {
  'my_first_db': 'postgres://...',
  'my_first_db_replica': 'postgres://...',
  'my_second_db': 'postgres://...',
  'my_second_db_replical': 'postgres://...'
}
```

And then in your model declarations:

```
class Db1Model(jardin.Model):
  db_name = {'master': 'my_first_db', 'replica': 'my_first_db_replica'}

class Db2Model(jardin.Model):
  db_name = {'master': 'my_second_db', 'replica': 'my_second_db_replica'}

class User(Db1Model): pass

class Project(Db2Model): pass
```

4.5 Replica lag measurement

You can measure the current replica lag in seconds using any class inheriting from `jardin.Model`:

```
jardin.Model.replica_lag()
# 0.001

MyModel.replica_lag()
# 0.001
```

4.6 Connection drops recovery

The exceptions `psycopg2.InterfaceError` and `psycopg2.OperationalError` are rescued and a new connection is initiated. Three attempts with exponential decay are made before bubbling up the exception.

CHAPTER 5

API reference

5.1 jardin.Model

class jardin.**Model**(***kwargs*)

Base class from which your models should inherit.

collection_class

alias of *Collection*

destroy(*force=False*)

Deletes the record. If the model has `soft_delete` activated, the record will not actually be deleted.

Parameters **force** (*boolean*) – forces the record to be actually deleted if `soft_delete` is activated.

classmethod find(*id*, ***kwargs*)

Finds a record by its id in the model's table in the replica database. :returns: an instance of the model.

classmethod find_by(*values={}*, ***kwargs*)

Returns a single record matching the criteria in `values` found in the model's table in the replica database.

Parameters **values** (*dict*) – Criteria to find the record.

Returns an instance of the model.

classmethod insert(***kwargs*)

Performs an INSERT statement on the model's table in the master database.

Parameters **values** (*dict*) – A dictionary containing the values to be inserted. `datetime`, `dict` and `bool` objects can be passed as is and will be correctly serialized by psycopg2.

classmethod last(*limit=1*, ***kwargs*)

Returns the last *limit* records inserted in the model's table in the replica database. Rows are sorted by `created_at`.

classmethod query(*sql=None*, *filename=None*, ***kwargs*)

run raw sql from sql or file against.

Parameters

- **sql** (*string*) – Raw SQL query to pass directly to the connection.
- **filename** (*string*) – Path to a file containing a SQL query. The path should be relative to CWD.
- **db** (*string*) – *optional* Database name from your `jardin_conf.py`, overrides the default database set in the model declaration.
- **role** (*string*) – *optional* One of `('master', 'replica')` to override the default.

Returns `jardin.Collection` collection, which is a `pandas.DataFrame`.

classmethod replica_lag(***kwargs*)
Returns the current replication lag in seconds between the master and replica databases.

Returns float

classmethod table_schema()
Returns the table schema.

Returns dict

classmethod transaction()
Enables multiple statements to be ran within a single transaction, see *Features*.

5.2 jardin.Collection

class jardin.**Collection**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)
Base class for collection of records. Inherits from *pandas.DataFrame*.

index_by(*field*)
Returns a dict with a key for each value of *field* and the first record with that value as value. :param field: Name of the field to index by. :type field: string.

records()
Returns an iterator to loop over the rows, each being an instance of the model's record class, i.e. jardin_record by default.

Index